START OF A NEW SERIES

Games with GO

ROLAND WADDILOVE sets off on a tour of techniques that will easily transform your machine code games

IN this series we're going to be looking at the techniques used in machine code games: Sprite print routines, collision detection, animation and coping with large numbers of sprites.

We'll build up a library of useful routines that you can incorporate into your own games and there will be several programs demonstrating their use.

They will all run on the BBC Micro, BBC B+, Master series and Electron, Basic I, Basic II, ADFS and DFS – so it doesn't matter what you've got.

Regular readers of *The Micro User* will recall Kevin Edwards' excellent series on writing machine code games which ran from February to September 1985.

I'm going to continue where Kevin left off, and assume that you've read his articles and typed in the listings. If you haven't then I suggest that you order the back issues immediately.

You'll certainly need the sprite designer from the June 1985 issue if you want to try out your own ideas. The programs are all available for downloading from Microlink.

Kevin's series discussed the Mode 2 memory map and covered simple sprite routines, reading the keyboard, generating random numbers and high score tables – the building blocks of machine code games.

To start my series I have combined several of the techniques already discussed with a few extra features, so some of it will be familiar and some will be new.

First of all enter the listing, run it and see what happens – if you've entered it correctly a coloured ball will appear.

It's fairly simple as machine code games go but there's a lot to learn from it. We'll break it down into simple sections and see what is going on.

MACHINE CODE GAMES

Part 1

Notice that the sprite is animated and apppears to rotate about a vertical axis. It is also under cursor control so press the cursor keys to move it round the screen.

You'll find that you can't move it out of the box in the centre of the screen.

And if you move it through the text you'll notice that it moves behind the first word, in front of the second and behind the third, just like the hardware sprites found on Atari and Commodore micros.

The sprite print routine labelled *print* starts at line 740 and uses the EOR method. If you followed the earlier series you will have seen several variations of this before.

Designed with animation in mind it runs from left to right printing each column of the sprite by EORing the data with the screen memory.

An animated sprite consists of several "frames" which are printed one after the other with each new frame being slightly different from the previous one – rather like a cartoon made up of separate drawings.

The print routine prints the new frame at the new address. At the same time the old frame is being erased from the old address.

The print routine expects the screen address of the old frame in *old* and the address of the new frame in *new*.

The old data address should be in *olddata+1* and the new data in *newdata+1*. The size is passed in *X* and *Y* where *X* is the number of columns and *Y* the number of rows.

To place a sprite on the screen initially the old address and data are unimportant because there's no previous frame to erase. Set *new* and *newdata* then call *put* instead of *print*. This puts a dummy address in *old*.

The frame number is stored in *frame* and is used to index into the character data table at line 1020. The address of the data for the first frame is stored at *data*, the next is at *data+2*, then *data+4* and so on.

There are four frames altogether and *frame* is incremented by two each time since each entry in the data table is two bytes long. You can see this in line 400.

To simplify the programming the sprite's position is stored as coordinates in *x%* and *y%*. The print routine requires the screen address so a call is made to *convert*.

This is a modified version of the

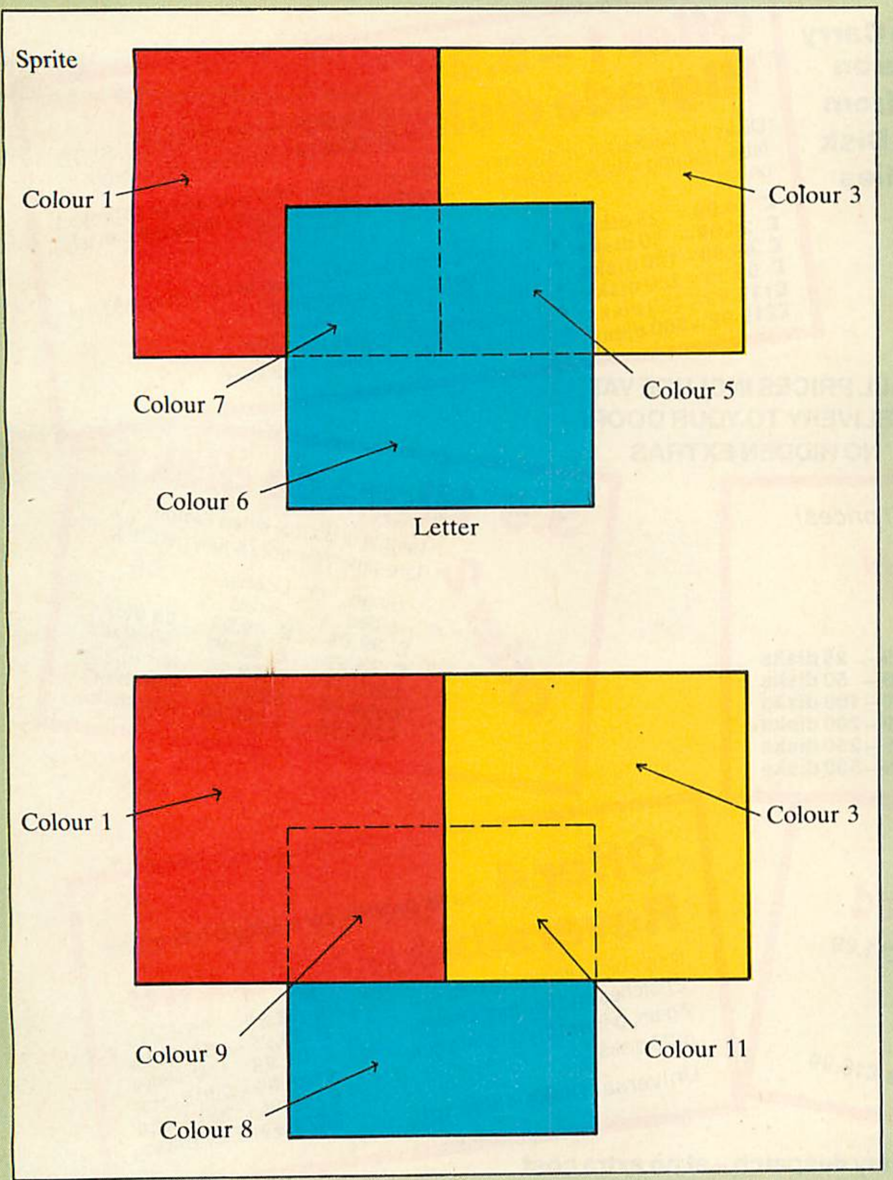


Figure I: The palette

convert routine used in the first series.

I've changed it so that it no longer relies on the OS rom for the 640 times table, but indexes *table* at the end of the code which stores the address of each screen line.

Any routine which relies on tables in the rom is likely to fail if Acorn decides to change it. This is one of the (many) reasons why some programs will not run properly on the Master series.

The program uses macros to save typing. A macro is a section of code which is given a name: When that name appears in the program the whole of the macro code is inserted.

For example line 520 is:

```
OPT FNinkey(-122)
```

which is used to test whether the cursor right key is pressed and is just like Basic's INKEY(–122). Notice how simple it is.

When the assembler comes to this line it stops assembling and evaluates the function *FNinkey* at the end of the program.

This re-enters the assembler, inserts the code and exits setting the assembly option to *pass,* which is what it was anyway. The assembler then carries on assembling .

Notice that the sprite's movement is restricted to the box in the centre of the screen. It needn't be in the centre – we could set up a sprite window anywhere on the screen.

Each of the subroutines *right, left, up* and *down* check that the sprite is within the limits set out by the box.

For instance in *right,* line 530 compares the *x* coordinate with 68 and if it is equal won't let you move any further. Similarly when going left you can't go less that 7.

One other important aspect is the way the sprite moves in front of and behind words like real hardware sprites. This is achieved by redefining the pallette. Take a look at Figure I to see how it's done.

The first word is printed in colour 6 and the sprite in colours 1 and 3. The sprite is EORed onto the screen. Now 6 EOR 1 is 7 and 6 EOR 3 is 5, so any red part of the spite on a letter will appear in colour 7 and any yellow part in colour 5.

If we set logical colours 7 and 5 to produce actual colour 6, the same as the letter, where the sprite and letter overlap appears in the actual colour of the letter. That is, the letter appears intact and the sprite will appear behind it.

Similarly the second word is printed in colour 8 which when EORed with 1 and 3 produce colours 9 and 11. These are set to produce actual colours 1 and 3, the same colours as the sprite.

Hence the overlapping parts appear in the colour of the sprite, causing it to appear to move in front of the letters.

I suppose you could call this method cheating since we aren't actually moving in front or behind anything and it does use up rather a lot of colours. However, it is quite effective and adds depth to the graphic display.

● *There is rather a lot to digest here but it's all fairly straightforward and you've got a whole month to experiment with the routines. Next month we'll be discussing collision detection methods.*

Full listing starts on Page 132

From Page 47

```
10 REM Animated Sprites
20 REM By R.A.Waddilove
30 REM (c) Micro User
40 RESTORE 1450:FOR I%=0 TO 191:READ J%:I%?&C00=J%:NEXT
50 PROCassemble
60 MODE 2:*FX16
70 VDU 23;8202;0;0;0;
80 VDU 19,7,6;0;
90 VDU 19,5,6;0;
100 VDU 19,8,6;0;
110 VDU 19,9,1;0;
120 VDU 19,11,3;0;
130 GCOL 0,2:DRAW 0,1023:DRAW 1278,1023:DRAW 1278,0:DRAW 0,0
140 GCOL 0,4:MOVE 96,120:PLOT 17,0,780:PLOT 17,1068,0:PLOT 17,0,-780:PLOT 17,-1068,0
150 COLOUR 6:PRINT TAB(2,10)"Use ";:COLOUR 8:PRINT "cursor ";:COLOUR 6:PRINT "keys!"
160 CALL &900
170 END
180
190 DEF PROCassemble
200 old=&70:new=&72:rows=&74:columns=&75:temp=&76:temp1=&78:temprows=&7A
210 address=&80:x%=&82:y%=&83:frame=&84
220 osbyte=&FFF4
230 FOR pass=0 TO 2 STEP 2
240 P%=&900
250 [ OPT pass
260
270 .initialise
280 LDA #&80:STA old:STA old+1  \set old address
290 LDX #35:STX x%:LDY #128:STY y%  \set your x,y
300 JSR convert  \get address
310 LDA new:STA address:LDA new+1:STA address+1  \save address
320 LDA data:STA newdata+1:LDA data+1:STA newdata+2  \set data
330 LDA #0:STA frame  \frame of animation
340 LDX #4:LDY #12:JSR print
350
360 .start
370 LDA #19:JSR osbyte
380 LDY frame
390 LDA data,Y:STA olddata+1:LDA data+1,Y:STA olddata+2  \set old data
400 TYA:CLC:ADC #2:AND #7:TAY:STY frame  \next frame
410 LDA data,Y:STA newdata+1:LDA data+1,Y:STA newdata+2  \set new data
420 LDA address:STA old:LDA address+1:STA old+1  \set old
430 JSR readkeys
440 LDX x%:LDY y%:JSR convert  \get new address
450 LDA new:STA address:LDA new+1:STA address+1  \save address
460 LDA &240:.fx19 CLI:SEI:CMP &240:BEQ fx19
470 LDX #4:LDY #12:JSR print
480 LDA #&81:LDX #&8F:LDY #&FF:JSR osbyte:TYA:BEQ start  \Escape pressed?
490 RTS  \return to Basic
500
510 .readkeys  \right
520 OPT FNinkey(-122):BEQ left
530 LDA x%:CMP #68:BEQ notright:INC x%
540 .notright
550 RTS
560 .left
570 OPT FNinkey(-26):BEQ up
580 LDA x%:CMP #7:BEQ notleft:DEC x%
590 .notleft
600 RTS
610 .up
620 OPT FNinkey(-58):BEQ down
630 LDA y%:CMP #34:BEQ notup:DEC y%:DEC y%
640 .notup
650 RTS
660 .down
670 OPT FNinkey(-42):BEQ notdown
680 LDA y%:CMP #212:BEQ notdown:INC y%:INC y%
690 .notdown
700 RTS
710
720 .put
730 LDA #&80:STA old:STA old+1
740 .print  \uses new/old/X=rows/Y=columns/olddata/newdata
750 STX columns:STY rows
760 LDX #0:LDY #0
770 LDA new:STA temp1:LDA new+1:STA temp1+1
780 LDA old:STA temp:LDA old+1:STA temp+1  \save address of column
790 .loop1
800 LDA rows:STA temprows
810 .loop2
820 .newdata LDA &3000,X:EOR (new),Y:STA (new),Y
830 .olddata LDA &3000,X:EOR (old),Y:STA (old),Y
840 INX:BNE noinc:INC olddata+2:INC newdata+2  \next data byte
850 .noinc
860 LDA old:AND #7:CMP #7:BEQ bottom1
870 INC old:BNE next1:INC old+1:JMP next1
880 .bottom1 \row
890 CLC:LDA old:ADC #&79:STA old:LDA old+1:ADC #2:STA old+1
900 .next1
910 LDA new:AND #7:CMP #7:BEQ bottom2
920 INC new:BNE next2:INC new+1:JMP next2
930 .bottom2
940 CLC:LDA new:ADC #&79:STA new:LDA new+1:ADC #2:STA new+1
950 .next2
960 DEC temprows:BNE loop2  \next row
970 CLC:LDA temp1:ADC #8:STA new:STA temp1:LDA temp1+1:ADC #0:STA new+1:STA temp1+1
980 LDA temp:ADC #8:STA old:STA temp:LDA temp+1:ADC #0:STA old+1:STA temp+1
990 DEC columns:BNE loop1  \next column
1000 RTS
1010
1020 .data  \character data
1030 OPT FNequw(&C00)
1040 OPT FNequw(&C00+48)
1050 OPT FNequw(&C00+2*48)
1060 OPT FNequw(&C00+3*48)
1070
1080 .convert   \X,Y -> address in new
1090 LDA #0:STA new+1:TXA:ASL A:ASL A:ROL new+1:ASL A:ROL new+1:STA new \X*8
1100 TYA:AND #7:ADC new:STA new:LDA new+1:ADC #0:STA new+1 \ +(Y MOD 8)
1110 TYA:LSR A:LSR A:LSR A:ASL A:TAY  \2*(Y DIV 8)
1120 LDA table,Y:ADC new:STA new:LDA table+1,Y:ADC new+1:STA new+1
1130 RTS
1140
1150 .table
1160 OPT FNtable
```

```
1170 ]
1180 NEXT
1190 ENDPROC
1200
1210 DEF FNtable
1220 FOR I%=0 TO 31
1230 ?P%=(&3000+I%*&280)MOD256
1240 P%?1=(&3000+I%*&280)DIV256
1250 P%=P%+2
1260 NEXT
1270 =pass
1280
1290 DEF FNequw(word)
1300 ?P%=word MOD256
1310 P%?1=word DIV256
1320 P%=P%+2
1330 =pass
1340
1350 DEF FNinkey(number)
1360 [ OPT pass
1370 LDA #&81
1380 LDY #&FF
1390 LDX #number+256
1400 JSR osbyte
1410 TYA
1420 ]
1430 =pass
1440
1450 REM BALL0
1460 REM X=4/Y=12
```

This is one of hundreds of programs now available FREE for downloading on MicroLink

```
 1470 DATA 0,1,1,3,15,7,7,15,3,1,1,0,
3,3,3,3,15,15,15,15,3,3,3,3,7,3,3,3,1
5,7,7,15,3,3,3,7,0,2,2,3,15,15,15,15,
3,2,2,0
 1480 REM BALL1
 1490 REM X=4/Y=12
 1500 DATA 0,1,1,3,15,11,11,15,3,1,1,
0,3,3,3,3,15,15,15,15,3,3,3,3,11,3,3,
3,15,11,11,15,3,3,3,11,0,2,2,3,15,15,
15,15,3,2,2,0
 1510 REM BALL2
 1520 REM X=4/Y=12
 1530 DATA 0,1,1,3,15,15,15,15,3,1,1,
0,7,3,3,3,15,7,7,15,3,3,3,7,3,3,3,3,1
5,15,15,15,3,3,3,3,0,2,2,3,15,7,7,15,
3,2,2,0
 1540 REM BALL3
 1550 REM X=4/Y=12
 1560 DATA 0,1,1,3,15,15,15,15,3,1,1,
0,11,3,3,3,15,11,11,15,3,3,3,11,3,3,3
,3,15,15,15,15,3,3,3,3,0,2,2,3,15,11,
11,15,3,2,2,0
```

This listing is included in this month's cassette tape offer. See order form on Page 167.

A crash course in collision detection

ROLAND WADDILOVE reveals two common methods of collision detection – PEEK and EOR

IN this series we are developing some of the basic techniques involved when writing machine code games.

These are not beginners articles and I'm assuming you have read Kevin Edwards' excellent introduction to machine code games in the earlier issues of *The Micro User* that I mentioned last month.

In that article we looked at sprite animation and masking techniques. I showed how a sprite could be restricted to any portion of the screen and made to move in front of and behind objects.

Now we'll move on to collision detection and look at some of the most common techniques used in commercial software.

Enter and run this month's program. (If PAGE is greater than &1900 you'll have to relocate it to &1900 or less.) You'll see a menu with two options – PEEK and EOR. These are the two methods of collision detection we'll be discussing this month.

Press 1 or 2 and after a short pause the screen will clear and a ball will be printed on the screen just below the word BEEP!. You'll see a box drawn around the edge of the screen.

Using the cursor keys, move the ball and see what happens when you pass over the letters of the word or the box. You should hear a beep as the collision is detected.

Press Escape and re-run the program. Try both methods and notice the difference – peek does not always detect the collision but eor does.

Take a look at line 370 to see how these routines work – here's the code:

```
JSR collision
BEQ OK
LDA £7
JSR oswrch
.OK
```

The subroutine *collision* checks whether the ball has hit anything and returns with the zero flag set if everything is ok. If a collision has occurred Ascii 7 is output to acknowledge the fact.

In an arcade game you would probably jump to an explosion or the end of game routine. However, for this demonstration the beep will suffice.

Let's look at *collision* in detail and see how it works. Ignore the first line

MACHINE CODE GAMES

Part 2

– 500. This merely flushes the sound buffer so the beep doesn't last too long.

A macro – FNbumped – combined with conditional assembly is used to select the type of collision routine to assemble. A flag is set in line 90 just after the menu and this is used in line 1370 to select either the peek or eor assembly code.

Peek is the shortest routine and starts at line 1390. The ball's address is stored in *address* and the simplest collision routine possible would just peek this screen address and see if the contents of the byte is what it should be – zero in our case.

Peek goes one further and looks at the top left and top right corners of the ball with:

```
LDY £0:LDA (address),Y
LDY £32:ORA (address),Y
```

If either of these two bytes are non-zero then a collision has occurred.

The point to note is that this routine simply looks at two bytes of the sprite and completely misses the rest which may have hit something.

This makes it a rather poor detection method but it does have it's uses. When we come to look at bouncing sprites you'll see how it can be put to good effect.

You could easily alter this simple routine to look above, below, to the left and right of the ball but it still wouldn't be perfect. You can see it's limitations if you try the demonstration.

The alternative method is EOR. While being a much superior method it takes far more code and is much slower.

The code starts at line 1480 and if you compare it to *print* you'll see it's quite similar. In fact it's just a modified print routine.

We looked at *print* last month and saw that to print the sprite the data is Exclusively ORed on to the screen. To remove it, the data is again EORed with the screen.

This EOR print method relies on the fact that if you take any number and EOR it with itself the result is zero. So, to move the sprite the data is again EORed with the screen and the old image is removed.

The print routine stores the character data in the screen memory so, if we EOR the data again with the screen memory the result should be zero.

To make this a little simpler to understand take the following list of numbers – it could be sprite data stored in the screen memory using *print*:

```
12,45,32,56,10
```

Now EOR each number with itself:

```
12 EOR 12 = 0
45 EOR 45 = 0
32 EOR 32 = 0
56 EOR 56 = 0
10 EOR 10 = 0
```

and the result each time is zero.

The EOR collision routine Exclusively ORs each byte of sprite data with the screen memory (but it doesn't store the result back in the memory). If any of the bytes are non-zero a collision must have occurred. You can see this in line 1560:

```
LDA (old),Y
BEQ zero
EOR (new),Y
BEQ zero
RTS
.zero
```

The routine returns immediately a non-zero result is found and the zero flag is zero. If there is no collision the routine returns with the zero flag set.

The line contains an additional refinement in that it only checks non-zero data bytes. So the empty space around the ball is ignored.

Lines 1480 to 1500 store the address of the sprite data in *old*, its screen address in *new* and load X and Y with the size. The collision routine proper starts at line 1510 and is a general subroutine which can be used with any sprite.

● *That just about wraps it up for this month. However, it's not the end of the story as next month we'll look at a totally different collision detection method. This will not involve looking at the screen – in fact the sprites don't even have to be on the screen!*

```
10 REM Collision Detection
20 REM By R.A.Waddilove
30 REM (c) Micro User
40 MODE6:*TV0,1
50 FOR I%=0 TO 47
60 READ J%:I%?&C00=J%
70 NEXT
80 PRINT'"Press (1)PEEK  (2)EOR";
90 F%=GET-49:VDU7
100 PROCassemble:CLEAR
110 MODE 2:VDU 23;8202;0;0;0;
120 VDU19,4,3;0;19,6,1;0;
130 GCOL 0,5:MOVE 96,120:PLOT 17,0,
780:PLOT 17,1068,0:PLOT 17,0,-780:PLO
T 17,-1068,0
140 COLOUR7:PRINT TAB(7,15)"BEEP!"
150 CALL &900
160 END
170
180 DEF PROCassemble
190 old=&70:new=&72
200 rows=&74:columns=&75
210 temp=&76:temp1=&78:temprows=&7A
220 address=&80:x%=&82:y%=&83
230 osbyte=&FFF4:oswrch=&FFEE
240 FOR pass=0 TO 2 STEP 2
250 P%=&900
260 [ OPT pass
270
280 .initialise
290 LDA #&80:STA old:STA old+1  \se
t old address
300 LDX #35:STX x%:LDY #160:STY y%
 \set your x,y
310 JSR convert  \get address
320 LDA new:STA address:LDA new+1:S
TA address+1  \save address
330 LDA data:STA newdata+1:LDA data
+1:STA newdata+2  \set data
340 LDX #4:LDY #12:JSR print
350
360 .start
370 JSR collision:BEQ OK:LDA #7:JSR
 oswrch  \BEEP!
380 .OK
390 LDA data:STA olddata+1:STA newd
ata+1:LDA data+1:STA olddata+2:STA ne
wdata+2  \set data
400 LDA address:STA old:LDA address
+1:STA old+1  \set old
410 JSR readkeys
420 LDX x%:LDY y%:JSR convert  \get
 new address
430 LDA new:STA address:LDA new+1:S
TA address+1  \save address
440 LDA &240:.fx19 CLI:SEI:CMP &240
:BEQ fx19
450 LDX #4:LDY #12:JSR print
460 LDA #&81:LDX #&0F:LDY #&FF:JSR
osbyte:TYA:BEQ start  \Escape pressed
?
470 RTS  \return to Basic
480
490 .collision
500 LDA #21:LDX #7:JSR osbyte
510 OPT FNbumped
520
530 .readkeys  \right
540 OPT FNinkey(-122):BEQ left
550 INC x%:RTS
560 .left
570 OPT FNinkey(-26):BEQ up
580 DEC x%:RTS
590 .up
600 OPT FNinkey(-58):BEQ down
610 DEC y%:DEC y%:RTS
620 .down
630 OPT FNinkey(-42):BEQ nokey
640 INC y%:INC y%
650 .nokey RTS
660
670 .put
680 LDA #&80:STA old:STA old+1
690 .print  \uses new/old/X=rows/Y=
columns/olddata/newdata
700 STX columns:STY rows
710 LDX #0:LDY #0
720 LDA new:STA temp1:LDA new+1:STA
 temp1+1
730 LDA old:STA temp:LDA old+1:STA
temp+1  \save address of column
740 .loop1
750 LDA rows:STA temprows
760 .loop2
770 .newdata LDA &3000,X:EOR (new),
Y:STA (new),Y
780 .olddata LDA &3000,X:EOR (old),
Y:STA (old),Y
790 INX:BNE noinc:INC olddata+2:INC
 newdata+2  \next data byte
800 .noinc
810 LDA old:AND #7:CMP #7:BEQ botto
m1
820 INC old:BNE next1:INC old+1:JMP
 next1
830 .bottom1 \row
840 CLC:LDA old:ADC #&79:STA old:LD
A old+1:ADC #2:STA old+1
```

Turn to Page 138

From Page 47

```
850 .next1
860 LDA new:AND #7:CMP #7:BEQ bottom2
870 INC new:BNE next2:INC new+1:JMP next2
880 .bottom2
890 CLC:LDA new:ADC #&79:STA new:LDA new+1:ADC #2:STA new+1
900 .next2
910 DEC temprows:BNE loop2  \next row
920 CLC:LDA temp1:ADC #8:STA new:STA temp1:LDA temp1+1:ADC #0:STA new+1:STA temp1+1
930 LDA temp:ADC #8:STA old:STA temp:LDA temp+1:ADC #0:STA old+1:STA temp+1
940 DEC columns:BNE loop1  \next column
950 RTS
960
970 .data  \character data
980 OPT FNequw(&C00)
990
1000 .convert  \X,Y->address in new
1010 LDA #0:STA new+1:TXA:ASL A:ASL A:ROL new+1:ASL A:ROL new+1:STA new \X*8
1020 TYA:AND #7:ADC new:STA new:LDA new+1:ADC #0:STA new+1 \ +(Y MOD 8)
1030 TYA:LSR A:LSR A:LSR A:ASL A:TAY \2*(Y DIV 8)
1040 LDA table,Y:ADC new:STA new:LDA table+1,Y:ADC new+1:STA new+1
1050 RTS
1060
1070 .table
1080 OPT FNtable
1090 ]
1100 NEXT
1110 ENDPROC
1120
1130 DEF FNtable
1140 FOR I%=0 TO 31
1150 ?P%=(&3000+I%*&280)MOD256
1160 P%?1=(&3000+I%*&280)DIV256
1170 P%=P%+2
1180 NEXT
1190 =pass
1200
1210 DEF FNequw(word)
1220 ?P%=word MOD256
1230 P%?1=word DIV256
1240 P%=P%+2
1250 =pass
1260
1270 DEF FNinkey(number)
1280 [ OPT pass
1290 LDA #&81
1300 LDY #&FF
1310 LDX #number+256
1320 JSR osbyte
1330 TYA
1340 ]:=pass
1350
1360 DEF FNbumped
1370 IF F% GOTO1460
1380
1390 REM PEEK Method
1400 [OPT pass
1410 LDY #0:LDA (address),Y
1420 LDY #32:ORA (address),Y
1430 RTS
1440 ]:=pass
1450
1460 REM EOR Method
1470 [OPT pass
1480 LDA address:STA new:STA temp:LDA address+1:STA new+1:STA temp+1 \new+temp=address
1490 LDA data:STA old:LDA data+1:STA old+1 \old points to sprite data
1500 LDX #4:LDY #12  \size
1510 STX columns:STY rows
1520 LDY #0
1530 .loop1
1540 LDX rows
1550 .loop2
1560 LDA (old),Y:BEQ zero:EOR (new),Y:BEQ zero:RTS
1570 .zero
1580 INC old:BNE n2:INC old+1  \next data byte
1590 .n2
1600 LDA new:AND #7:CMP #7:BEQ botrow  \next screen byte
1610 INC new:BNE n1:INC new+1:BNE n1
1620 .botrow
1630 CLC:LDA new:ADC #&79:STA new:LDA new+1:ADC #2:STA new+1
1640 .n1
1650 DEX:BNE loop2  \next row
1660 LDA temp:ADC #8:STA new:STA temp:LDA temp+1:ADC #0:STA new+1:STA temp+1
1670 DEC columns:BNE loop1  \next column
1680 RTS
1690 ]:=pass
1700
1710 REM Ball data
1720 REM columns=4/rows=12
1730 DATA 0,1,1,3,15,7,7,15,3,1,1,0,3,3,3,3,15,15,15,15,3,3,3,3,7,3,3,3,15,7,7,15,3,3,3,7,0,2,2,3,15,15,15,15,3,2,2,0
```

This listing is included in this month's cassette tape offer. See order form on Page 159.

PROBLEM

I RECENTLY bought a Rediffusion system Alpha 14in colour monitor, for use with a BBC B. While mainly happy with it there appears to be a fault that no one can cure.

The fault is that very occasionally the left side of the screen is displayed on the right and vice versa. The display also goes very dim.

The fault only occurs when a new screen is displayed or a CLS or a change of mode – and as I said only occassionally.

The fault is cleared by cleaning the screen repeatedly – not guaranteed to work – or by switching the monitor off and on, which is guaranteed to work. – Doreen Edwards, Manchester.

SOLVED

When you change screen modes the line and frame syncronising signals are stopped and started again from the beginning. This means that the monitor has to re-syncronise itself.

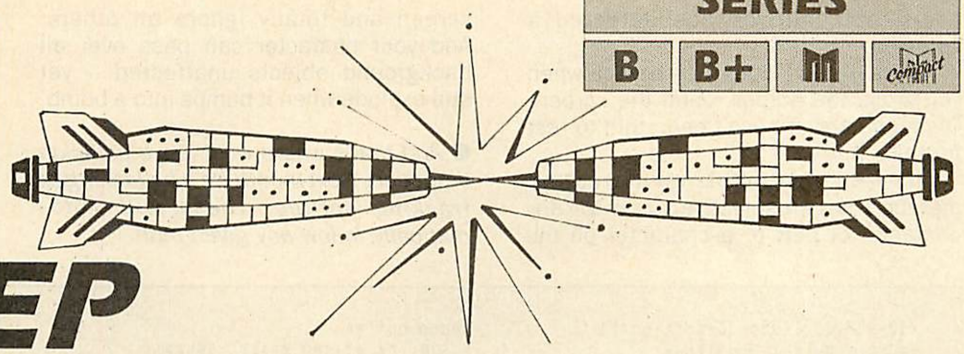
What is happening is that your monitor is re-syncronising out of phase with the line sync pulses. This is definitely a fault in the sync circuits of your monitor.

You could try adjusting the line sync control which you will find inside the set. If this is adjusted so that the natural line frequency is further away from the syncronising signals then maybe you will get better lock and this problem will not recur.

Roland Waddilove
looks at another
method of detecting
colliding sprites

SERIES

B **B+** **M** **compact**



BEEP marks the spot

LAST month we looked at two simple methods of detecting collisions between sprites – EOR and peek.

There are many different ways of doing this and machine code arcade games often use more than one depending on the circumstances. So this month I'm going to show you one more method and this is probably the best and most widely used.

The routine used is extremely fast, accurate and does not involve peeking the screen memory. In fact the sprites do not have to be on the screen at all

Figure I shows two simple rectangular sprites that have collided. Sprite one is on the left, sprite two on the right.

The coordinates of sprite one are *x1,y1* and its width and height are *w1* and *h1*. These parameters will be known as they are also used by the print routine *print*.

Similarly sprite two is at *x2,y2* and is *w2* bytes wide and *h2* bytes high.

Here's the algorithm used to test for collision:

IF x1 is less than x2 THEN add w1 to x1, see if this is greater than x2 and return if false ELSE add w2 to x2, see if this is greater than x1 and return if false.

IF y1 is less than y2 THEN add h1 to y1, see if this is greater than y2 and return ELSE add h2 to y2, see if this is greater than y1 and return.

MACHINE CODE GAMES

Part 3

In plain language, it finds out which sprite is further left, one or two. It adds the width to the x coordinate of the left sprite and sees whether this is greater than the x coordinate of the right sprite. If it isn't they can't possibly have collided.

Then it tests which is higher up the screen, sprite one or two. It adds the height to the top sprite's y coordinate and sees whether this is greater than the bottom sprite's y coordinate. If it is the sprites have definitely collided.

To see this in machine code take a look at the subroutine labelled *collision* in this month's listing. It's a general routine which tests the two sprites whose coordinates are stored in *x1,y1, x2,y2* with dimensions in *w1,h1, w2,h2*. It returns with the carry flag set if they have collided.

The subroutine *bumped* sets up these parameters and calls *collision*. If the carry is set Ascii 7 is output and you'll hear a beep. Of course it's up to you what happens and in a real arcade game you would probably have an explosion or some other suitable routine.

That's enough of the theory, it's time to see the routines in action so enter and run this month's listing.

You'll see two sprites, one in the centre of the screen and one at the bottom left. Use the cursor keys to move the bottom sprite and notice what happens when they collide – you'll hear a loud beep. This is *bumped* letting you

Turn to Page 112 ▶

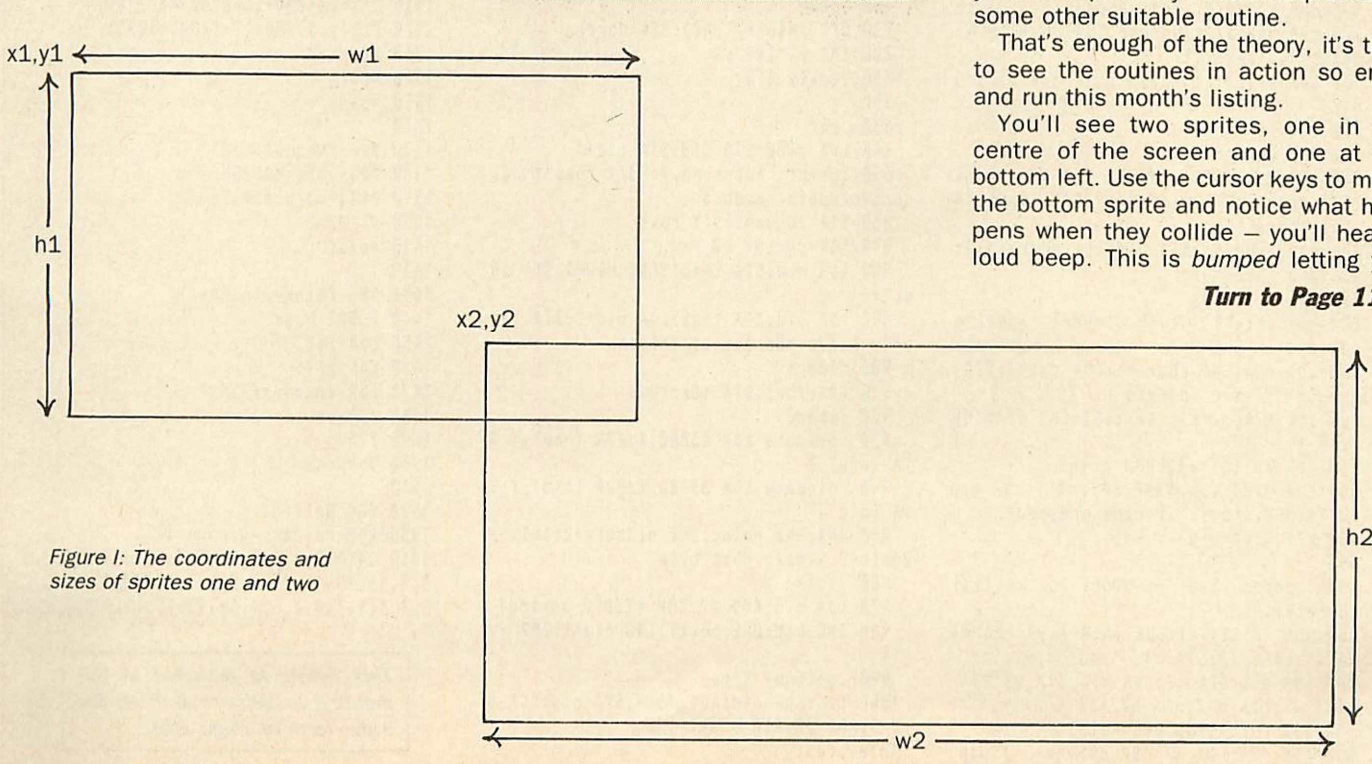


Figure I: The coordinates and sizes of sprites one and two

◀ From Page 111

know that *collision* has detected a collision.

Notice that it does not detect when you cross the border round the screen. Of course not, it hasn't been told to test for this.

So, with this collision detection method you can selectively test for any character or part of a character on the screen and totally ignore all others. And your character can pass over all background objects unaffected – yet still explode when it bumps into a bomb.

● *And that's where we'll leave collision detection. Next month we'll be looking at tracking sprites. These will automatically follow any given path.*

```
10 REM Collision Detection II
20 REM By R.A.Waddilove
30 REM (c) Micro User
40 MODE7:*TV0,1
50 FOR I%=0 TO 47
60 READ J%:I%?&C00=J%
70 NEXT
80 PROCassemble:CLEAR:*FX16
90 MODE 2:VDU 23;8202;0;0;0;
100 MOVE 100,100:DRAW 100,923:DRAW 1178,923:DRAW 1178,100:DRAW 100,100
110 CALL &900
120 END
130
140 DEF PROCassemble
150 old=&70:new=&72:rows=&74:columns=&75:temp=&76:temp1=&78:temprows=&7A
160 x1=&90:y1=&91:w1=&92:h1=&93:x2=&94:y2=&95:w2=&96:h2=&97
170 address=&80:x%=&82:y%=&83
180 alien=&84:ax%=&86:ay%=&87
190 osbyte=&FFF4:oswrch=&FFEE
200 FOR pass=0 TO 2 STEP 2
210 P%=&900
220 [ OPT pass
230
240 .initialise
250 LDA data:STA newdata+1:LDA data+1:STA newdata+2  \set data
260 LDX #35:STX ax%:LDY #128:STY ay% \set alien x,y
270 JSR convert  \get address
280 LDX #4:LDY #12:JSR put  \print alien
290 LDA data:STA newdata+1:LDA data+1:STA newdata+2  \set data
300 LDX #7:STX x%:LDY #216:STY y%  \set your x,y
310 JSR convert  \get address
320 LDA new:STA address:LDA new+1:STA address+1  \save it
330 LDX #4:LDY #12:JSR put  \print you
340
350 .start
360 JSR bumped
370 LDA data:STA olddata+1:STA newdata+1:LDA data+1:STA olddata+2:STA newdata+2  \set data
380 LDA address:STA old:LDA address+1:STA old+1  \set old
390 JSR readkeys
400 LDX x%:LDY y%:JSR convert  \get new address
410 LDA new:STA address:LDA new+1:STA address+1  \save address
420 LDA &240:.fx19 CLI:SEI:CMP &240:BEQ fx19
430 LDX #4:LDY #12:JSR print
440 LDA #&81:LDX #&8F:LDY #&FF:JSR osbyte:TYA:BEQ start  \Escape pressed?
450 RTS  \return to Basic
460
470 .bumped  \set up block for collision detection
480 LDA x%:STA x1:LDA y%:STA y1:LDA #4:STA w1:LDA #12:STA h1  \your x,y,size
490 LDA ax%:STA x2:LDA ay%:STA y2:LDA #4:STA w2:LDA #12:STA h2 \alien x,y,size
500 JSR collision:BCC nothit
510 LDA #21:LDX #7:JSR osbyte  \flush sound buffer
520 LDA #7:JMP &FFEE  \BEEP!
530 .nothit
540 RTS
550
560 .collision
570 LDA x1:CMP x2:BCC c1  \x1>x2 ?
580 LDA x2:CLC:ADC w2:CMP x1  \x2+w2>x1 ?
590 BCS checky:RTS
600 .c1 ADC w1:CMP x2  \x1+w1>x2 ?
610 BCS checky:RTS
620 .checky
630 LDA y1:CMP y2:BCC c2  \y1>y2 ?
640 LDA y2:CLC:ADC h2:CMP y1  \y2+h2>y1 ?
650 RTS
660 .c2 ADC h1:CMP y2  \y1+h1>y2 ?
670 RTS
680
690 .readkeys  \right
700 OPT FNinkey(-122):BEQ left
710 INC x%:RTS
720 .left
730 OPT FNinkey(-26):BEQ up
740 DEC x%:RTS
750 .up
760 OPT FNinkey(-58):BEQ down
770 DEC y%:DEC y%:RTS
780 .down
790 OPT FNinkey(-42):BEQ nokey
800 INC y%:INC y%
810 .nokey RTS
820
830 .put
840 LDA #&80:STA old:STA old+1
850 .print  \uses new/old/X=rows/Y=columns/olddata/newdata
860 STX columns:STY rows
870 LDX #0:LDY #0
880 LDA new:STA temp1:LDA new+1:STA temp1+1
890 LDA old:STA temp:LDA old+1:STA temp+1  \save address of column
900 .loop1
910 LDA rows:STA temprows
920 .loop2
930 .newdata LDA &3000,X:EOR (new),Y:STA (new),Y
940 .olddata LDA &3000,X:EOR (old),Y:STA (old),Y
950 INX:BNE noinc:INC olddata+2:INC newdata+2  \next data byte
960 .noinc
970 LDA old:AND #7:CMP #7:BEQ bottom1
980 INC old:BNE next1:INC old+1:JMP next1
990 .bottom1 \row
1000 CLC:LDA old:ADC #&79:STA old:LDA old+1:ADC #2:STA old+1
1010 .next1
1020 LDA new:AND #7:CMP #7:BEQ bottom2
1030 INC new:BNE next2:INC new+1:JMP next2
1040 .bottom2
1050 CLC:LDA new:ADC #&79:STA new:LDA new+1:ADC #2:STA new+1
1060 .next2
1070 DEC temprows:BNE loop2  \next row
1080 CLC:LDA temp1:ADC #8:STA new:STA temp1:LDA temp1+1:ADC #0:STA new+1:STA temp1+1
1090 LDA temp:ADC #8:STA old:STA temp:LDA temp+1:ADC #0:STA old+1:STA temp+1
1100 DEC columns:BNE loop1 \next column
1110 RTS
1120
1130 .data  \character data
1140 OPT FNequw(&C00)
1150
1160 .convert  \X,Y->address in new
1170 LDA #0:STA new+1:TXA:ASL A:ASL A:ROL new+1:ASL A:ROL new+1:STA new \X*8
1180 TYA:AND #7:ADC new:STA new:LDA new+1:ADC #0:STA new+1 \ +(Y MOD 8)
1190 TYA:LSR A:LSR A:LSR A:ASL A:TAY  \ 2*(Y DIV 8)
1200 LDA table,Y:ADC new:STA new:LDA table+1,Y:ADC new+1:STA new+1
1210 RTS
1220
1230 .table
1240 OPT FNtable
1250 ]
1260 NEXT
1270 ENDPROC
1280
1290 DEF FNtable
1300 FOR I%=0 TO 31
1310 ?P%=(&3000+I%*&280)MOD256
1320 P%?1=(&3000+I%*&280)DIV256
1330 P%=P%+2
1340 NEXT
1350 =pass
1360
1370 DEF FNequw(word)
1380 ?P%=word MOD256
1390 P%?1=word DIV256
1400 P%=P%+2
1410 =pass
1420
1430 DEF FNinkey(number)
1440 [ OPT pass
1450 LDA #&81
1460 LDY #&FF
1470 LDX #number+256
1480 JSR osbyte
1490 TYA
1500 ]:=pass
1510
1520 REM Ball data
1530 REM columns=4/rows=12
1540 DATA 0,1,1,3,15,7,7,15,3,1,1,0,3,3,3,3,15,15,15,15,3,3,3,3,7,3,3,3,15,7,7,15,3,3,3,7,0,2,2,3,15,15,15,15,3,2,2,0
```


This listing is included in this month's cassette tape offer. See order form on Page 159.

Roland Waddilove continues his series on machine code arcade games

SERIES

B B+ M Compact

On the track of sprites

MACHINE CODE GAMES

Part 4

IN the last two articles we looked at three different methods of detecting collisions between sprites. Now it is time to move on and this month we'll be seeing how to implement tracking sprites.

Once set in motion these characters will automatically follow a preset pattern round the screen.

You may be wondering why we should need such sprites when writing arcade games. Well, they are more common than you may think.

The ghosts in Pac Man follow a pattern around the maze, so do the aliens in Arcadians as they swoop down. Perhaps the best example of this class of sprites are the aliens in Galaforce – the game Kevin Edwards wrote for Superior Software last year.

In this game aliens stream in from off the edge of the screen, perform an amazing pattern and zoom off again, exploding as they reach the edge. And what is more, each new level brings a new wave of aliens with an even more intricate pattern.

The prospect of programming this may seem daunting, but as we'll see it is in fact fairly straightforward if you go about it in the right way.

Faced with a task like this it is vital that you plan your program and data very carefully. It's so easy to become entangled with spaghetti-like code with pages and pages of data.

Before you put finger to keyboard the structure of the pattern data must be worked out. I'm not talking about the data itself here, but the way it is to be stored in memory.

This must be done before any code has been entered into the micro – you can't write the program unless you know how the data is structured.

Of course, there's more than one way to skin a cat and I can't possibly go through all the possible data structures. Here is just one solution, using a single byte to specify each point in the pattern

Imagine a sprite currently at the coordinates x1,y1, following a pattern on the screen.

There are two ways to specify its next position, x2,y2. We can either provide the absolute coordinates to move to or give it the new coordinates relative to its current position.

Sprites normally move in small steps, often a byte at a time. So, the new coordinate relative to the current position will always be a small number.

If we restrict the relative movement to the range 0 to 3, 3 being ¾ of a normal mode 2 character's width, only two bits are required to store the offset. Remember that the movement can be positive or negative so an extra bit is necessary to hold the sign.

This means that the new coordinates can be stored in six bits, three for the x displacement and three for the y. Since there are eight bits in a byte this leaves two bits free.

One of the spare bits can determine whether a bomb is to drop or some other action is to occur and the remaining bit can be a GOTO bit – causing a jump to another part of the pattern.

Take a look at Figure I, which shows the structure of the pattern data. Bits 0 to 2 and 3 to 5 are the signed x and y coordinates relative to the current position.

If bit 6 is set a bomb is dropped at this point in the pattern and bit 7 means GOTO – the numbers stored in bits 0 to 6 are subtracted from the current position in the pattern.

For instance, a pattern byte of &85 would mean go back five bytes in the pattern. This enables a sprite to repeat a section, fly in circles or go back to the start.

There are certain values of the pattern byte that will never occur – 4, 32 and 36. In binary 4 is %00000100 indicating an x offset of –0, which is meaningless. Similarly –0 for y and –0 for both x and y should not arise.

These numbers can be treated as special cases. All indicate the end of the pattern in one way or another, 4 means the sprite is to explode at this point, 32 means remove it from the screen and 36 means turn it into a kamikazi sprite so it will make a beeline straight for you.

The two bytes at the beginning of a pattern hold the absolute x and y coordinates of the start position. The pattern itself starts at byte two and ends with one of the special cases outlined above or a GOTO.

As you can see it is possible to cram a mass of information into just one byte. This is vital on the BBC range of micros as there is only about 10k of ram to play around with in modes 0, 1 and 2.

Now we can move on to the data structure for the sprites themselves.

Each sprite requires a six byte block of memory – two for the absolute coordi-

Turn to Page 134 ▶

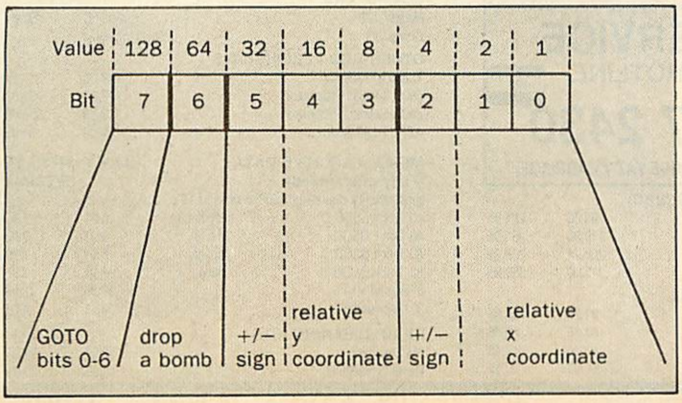


Figure I: The structure of the pattern byte

Byte Function
0 = flags
1 = pattern pointer
2 = x coordinate
3 = y coordinate
4/5 = screen address

Flags
Flags
bit 0 = on screen
bit 1 = exploding
bit 2 = dead
bit 3 = start exploding

Figure II: The sprite information block

◀ From Page 133

nates, two for the screen address, one for a pointer to the current position in the pattern and one for various flags. Figure II shows the sprite information block that I use.

Bit 0 of the flags indicates whether the sprite is on the screen. Bit 1 means it is exploding and bit 3 means start it exploding. Bit 2 means it is dead.

It is convenient to store all the blocks in one page of memory and index into them using the X and Y registers. This means that there can be up to 256/6 or 42 tracking sprites on the screen.

It's now time to put the theory into practice so enter and save this month's listing.

When you run it you'll first be asked for the spacing between the sprites. Try 7 to start off. At the next prompt enter the number of sprites, remembering that you can only have 42 at most.

The screen will clear and you'll see a stream of aliens flying in from the top left corner of the screen. They'll loop round the bottom and zoom off, disappearing at the top right of the screen.

Press Escape and run it again, experimenting with the spacing and number of sprites.

The structure of the data makes the programming less difficult than you might first have thought, having just seen the demonstration.

There are several points to watch out for when writing this sort of routine. Firstly all the sprites are off the screen when the program is first run and must be printed at the start position one at a time.

Only those sprites on the screen are moved and these are recognised by having bit zero of their flags byte set.

When putting sprites on the screen it's important not to confuse those that have finished the pattern and have been taken off the screen. These are recognised by having bit 2 of their flags byte set.

Only one sprite at a time can be printed at the start position and before the next one is printed the last one must have moved out of the way.

This is the spacing between the sprites. Try the demonstration with a spacing of one and you'll see the problems that can occur.

The solution is to have a timer. This counts down the next sprite to be put on the screen.

Finally, try designing your own patterns and replace the data statements at the end of the listing with your own. Take note of how the data is structured and it's up to you to make sure that the sprites stay within the confines of the screen limits.

● *That's all for now. There is plenty here to get your teeth into and it should keep you busy until next month when we'll move on to bouncing sprites.*

```
   10 REM Tracking Sprites
   20 REM By R.A.Waddilove
   30 REM (c) Micro User
   40 MODE 7:*TV0,1
   50 IF PAGE<>&1900 PRINT"PAGE must be
&1900":STOP
   60 RESTORE 1730:FOR I%=0 TO 47:READ J
%:I%?&C00=J%:NEXT
   70 pattern=&900:balls=&A00
   80 RESTORE 1760:I%=0:REPEAT:READ J%:I
%?pattern=J%:I%=I%+1:UNTIL J%<0
   90 INPUT'"Space between balls(1-20):"
space
  100 INPUT'"Number of balls(1-40):"numb
er-of-balls
  110 PROCassemble:CLEAR
  120 MODE 2:CALL &1100
  130 END
  140
  150 DEF PROCassemble
  160 old=&70:new=&72:rows=&74:columns=&
75:temp=&76:temp1=&78
  170 delay=&7B
```

Turn to Page 137 ▶

◀ *From Page 134*

```
 180 ball-at-start=&7C
 190 ball-number=&7D
 200 osbyte=&FFF4
 210 FOR pass=0 TO 2 STEP 2
 220 P%=&1100
 230 [ OPT pass
 240 .initialise
 250 LDA #space:STA delay
 260 LDA #0:TAX
 270 .loop
 280 STA balls,X  \initialise balls
 290 DEX:BNE loop
 300 .main-loop
 310 JSR move-balls
 320 DEC delay:BPL not-time
 330 LDA #space:STA delay
 340 .not-time
 350 LDA #&81:LDX #256-113:LDY #&FF:JSR
osbyte  \Escape?
 360 TYA:BEQ main-loop
 370 RTS
 380
 390 .move-balls
 400 LDA #0:STA ball-at-start  \one new
ball at a time
 410 LDX #number-of-balls*6
 420 LDA &240:.fx19 CLI:SEI:CMP &240:BE
Q fx19
 430 .mbloop
 440 STX ball-number
 450 LDA balls,X:AND #1:BNE on-screen
 460 JSR new-ball:JMP next-ball
 470 .on-screen
 480 LDA balls,X:AND #10:BEQ move-it
 490 JSR explode:JMP next-ball
 500 .move-it
 510 LDA data:STA olddata+1:STA newdata
+1:LDA data+1:STA olddata+2:STA newdata+
2
 520 LDA balls+4,X:STA old:LDA balls+5,
X:STA old+1 \old address
 530 JSR get-new-xy
 540 LDY balls+3,X:LDA balls+2,X:TAX:JS
R convert \new address
 550 LDX ball-number
 560 LDA new:STA balls+4,X:LDA new+1:ST
A balls+5,X  \store it
 570 JSR print
 580 .next-ball
 590 LDA ball-number:SEC:SBC #6:TAX
 600 BNE mbloop
 610 CLI
 620 RTS
 630
 640 .get-new-xy
 650 LDY balls+1,X:INC balls+1,X  \patt
ern index
 660 LDA pattern,Y
 670 CMP #32:BNE ex1:JMP remove
 680 .ex1
 690 CMP #4:BNE ex2:LDA #9:STA balls,X:
RTS  \explode next time
 700 .ex2
 710 CMP #36:BNE moving:JMP kamikasi
 720 .moving
 730 LDA pattern,Y:AND #4:BEQ xplus  \l
eft or right?
 740 LDA pattern,Y:AND #3:STA temp:SEC:
LDA balls+2,X:SBC temp:JMP got-x
 750 .xplus
 760 LDA pattern,Y:AND #3:CLC:ADC balls
+2,X
 770 .got-x
 780 STA balls+2,X
 790 LDA pattern,Y:AND #32:BEQ yplus  \
up or down?
 800 LDA pattern,Y:LSR A:LSR A:AND #6:S
TA temp:SEC:LDA balls+3,X:SBC temp:JMP g
ot-y
 810 .yplus
 820 LDA pattern,Y:LSR A:LSR A:AND #6:C
LC:ADC balls+3,X
 830 .got-y
 840 STA balls+3,X
 850 RTS
 860
 870 .remove
 880 LDA #4:STA balls,X  \dead
 890 LDA &80:STA new:STA new+1:JSR prin
t \off screen
 900 PLA:PLA:JMP next-ball
 910
 920 .explode
 930 \Write it yourself!
 940 RTS
 950
 960 .kamikasi
 970 \Write it yourself!
 980 RTS
 990
 1000 .new-ball
 1010 LDA delay:BNE no-ball
 1020 LDA ball-at-start:BNE no-ball
 1030 LDA balls,X:AND #4:BNE no-ball  \d
ead?
 1040 INC ball-at-start  \no more
 1050 LDA #1:STA balls,X  \on screen
```



```
 1060 LDA #2:STA balls+1,X  \pattern sta
rt
 1070 LDA pattern+1:STA balls+3,X:TAY:LD
A pattern:STA balls+2,X:TAX \start x,y
 1080 JSR convert
 1090 LDX ball-number
 1100 LDA new:STA balls+4,X:LDA new+1:ST
A balls+5,X  \store address
 1110 LDA data:STA newdata+1:LDA data+1:
STA newdata+2
 1120 JMP put
 1130 .no-ball
 1140 RTS
 1150
 1160 .put
 1170 LDA #&80:STA old:STA old+1
 1180 .print
 1190 LDA #4:STA columns
 1200 LDX #0:LDY #0
 1210 LDA new:STA temp1:LDA new+1:STA te
mp1+1
 1220 LDA old:STA temp:LDA old+1:STA tem
p+1
 1230 .loop1
 1240 LDA #12:STA rows
 1250 .loop2
 1260 .newdata LDA &3000,X:EOR (new),Y:S
TA (new),Y
 1270 .olddata LDA &3000,X:EOR (old),Y:S
TA (old),Y
 1280 INX
 1290 LDA old:AND #7:CMP #7:BEQ bottom1
 1300 INC old:BNE next1:INC old+1:JMP ne
xt1
 1310 .bottom1
 1320 CLC:LDA old:ADC #&79:STA old:LDA o
ld+1:ADC #2:STA old+1
 1330 .next1
 1340 LDA new:AND #7:CMP #7:BEQ bottom2
 1350 INC new:BNE next2:INC new+1:JMP ne
xt2
 1360 .bottom2
 1370 CLC:LDA new:ADC #&79:STA new:LDA n
ew+1:ADC #2:STA new+1
 1380 .next2
 1390 DEC rows:BNE loop2
 1400 CLC:LDA temp1:ADC #8:STA new:STA t
emp1:LDA temp1+1:ADC #0:STA new+1:STA te
mp1+1
 1410 LDA temp:ADC #8:STA old:STA temp:L
DA temp+1:ADC #0:STA old+1:STA temp+1
 1420 DEC columns:BNE loop1
 1430 RTS
 1440
 1450 .data OPT FNequw(&C00)
 1460
 1470 .convert   \X,Y -> address
 1480 LDA #0:STA new+1:TXA:ASL A:ASL A:R
OL new+1:ASL A:ROL new+1:STA new \X*8
 1490 TYA:AND #7:ADC new:STA new:LDA new
+1:ADC #0:STA new+1 \ +(Y MOD 8)
 1500 TYA:LSR A:LSR A:LSR A:ASL A:TAY  \
2*(Y DIV 8)
 1510 LDA table,Y:ADC new:STA new:LDA ta
ble+1,Y:ADC new+1:STA new+1
 1520 RTS
 1530
 1540 .table OPT FNtable
 1550 ]
 1560 NEXT
 1570 ENDPROC
 1580
 1590 DEF FNequw(word)
 1600 ?P%=word MOD256
 1610 P%?1=word DIV256
 1620 P%=P%+2
 1630 =pass
 1640
 1650 DEF FNtable
 1660 FOR I%=0 TO 31
 1670 ?P%=(&3000+I%*&280)MOD256
 1680 P%?1=(&3000+I%*&280)DIV256
 1690 P%=P%+2
 1700 NEXT
 1710 =pass
 1720
 1730 REM Ball data...X=4/Y=12
 1740 DATA 0,1,1,3,15,7,7,15,3,1,1,0,3,3
,3,15,15,15,15,3,3,3,7,3,3,3,15,7,7,
15,3,3,3,7,0,2,2,3,15,15,15,15,3,2,2,0
 1750 REM ******* Pattern Data *******
 1760 DATA 0,8
 1770 DATA 27,27,27,27,27,27,27,27,27
 1780 DATA 27,27,27,27,27,26,26,26,26
 1790 DATA 17,17,17,17,17,17,17,17,17
 1800 DATA 16,16,16,16,8,8,8,8,8,8
 1810 DATA 5,5,5,6,6,6,7,7,7,7,7,7
 1820 DATA 6,6,6,5,5,5,40,40,40,40
 1830 DATA 40,40,48,48,48,48,49,49,49
 1840 DATA 49,49,49,49,49,49,58,58,58
 1850 DATA 58,59,59,59,59,59,59,59,59
 1860 DATA 59,59,59,59
 1870 DATA 32,-1
```

This listing is included in this month's cassette tape offer. See order form on Page 159.

Roland Waddilove concludes his series on writing machine code arcade games

SERIES

B B+ M

Let's bounce a sprite

MACHINE CODE GAMES Part 5

WE have in this series been developing the routines and techniques required for writing machine code arcade games. We've discussed animation, collision detection and tracking sprites which follow a set pattern.

Now it is time to move on to bouncing sprites.

These are commonly found in multi-screen ladders and levels-type arcade games, bouncing between walls, climbing up and down ladders, running back and forth along the platforms and so on.

A window can be defined for each bouncer, and the sprites will automatically bounce within them. They may move horizontally, vertically, in all directions or stay still. Each sprite may be animated and different from the others in size and shape.

To see bouncing sprites in action enter and run this month's program. You'll see four boxes on the screen, each containing a ball.

One bounces in all directions, one left and right off the walls, one up and down. The last remains where it is.

The walls are unimportant, and are there only to define the windows' edges. If you erase them the balls will still bounce in the correct positions.

Before writing a machine code routine to perform this sort of animation it is essential to structure the data. Once this is done the programming becomes much simpler.

Figure I shows the data structure. Each bouncing sprite requires a 20 byte information block, and all must fit in one page of memory for easy indexing with the X register.

The first four items in the block, bytes 0 to 7, are pointers to the sprite data for four frames of animation. You don't have to have four-frame animation, and could, in fact, make them all the same – the facility is available if required.

This enables each bouncer to be different from the others: Tracking sprites, as we saw last month, are all the same.

Byte 8 stores the current frame of animation, which, incidentally, runs 0, 2, 4, 6 and not 1, 2, 3, 4.

Byte 9 stores various flags – seen in Figure II. Bit 0 shows the horizontal direction, left or right, bit 2 the vertical direction, up or down. Bit 1 or bit 3 – if zero – means don't bother moving it.

The effect is that if bits 1 and 3 are set the sprite will bounce in all directions. If only bit 1 is set it will bounce left and right, and if bit 3 is set it will go up and down.

Of course if they are both zero the sprite stays put, though it is still animated.

Bit 7 of the flags shows whether the character is on screen.

Bytes 10 and 11 of the information block store the current address, and 12 and 13 hold the x and y coordinates. Don't forget that the screen is 80 bytes wide and 256 deep.

Bytes 14 and 15 store the sprite's size in columns and rows. Each sprite may be the same or different from the others. I've used the same sprite throughout to save you entering masses of data.

Bytes 16 to 19 set the left, right, top and bottom edges of the bouncer's window. The sprites move vertically in steps of two (to even out the fine vertical but coarse horizontal resolution of the screen) so the top and bottom edges and the y coordinate must be all odd or all even numbers.

You must set up each bouncer information block correctly before calling the machine code. You can see this in PROCbouncer_table. It reads four lines of data and stores them in page 9.

To change the sprites, windows, direction and animation all you need to change are four data statements. This simplicity enables you to have large numbers of screens in your game.

If you've followed this series from the start you should by now have all the routines you need to write your first arcade game.

The machine code is structured and the subroutines are quite general, so they can be extracted easily and inserted into other programs.

● *And that's the end of this series. I haven't covered every possible machine code games technique, in fact I've barely scratched the surface, but we have covered enough to get you started. Have fun exploring further!*

Byte	Function
0	data0
2	data1
4	data2
6	data3
8	frame
9	flags
10	address
12	x
13	y
14	columns
15	rows
16	min x
17	max x
18	min y
19	max y

Figure I: Structure of the bouncer information block

Bit	Function
0	1=move left, 0=move right
1	1=move horizontally, 0=leave alone
2	1=move up, 0=move down
3	1=move vertically, 0=leave alone
4	Not used
5	Not used
6	Not used
7	1=On screen

Figure II: Structure of the flags byte

```
   10 REM Bouncing Sprites
   20 REM By R.A.Waddilove
   30 REM (c) Micro User
   40 MODE 7:*TV255,1
   50 RESTORE 1720:FOR I%=0 TO 191:READ
J%:I%?&C00=J%:NEXT
   60 PROCbouncer-table
   70 PROCassemble:CLEAR
   80 MODE 2:*FX16
   90 VDU 23;8202;0;0;0;
  100 DRAW 0,1023:DRAW 1278,1023
  110 DRAW 1278,0:DRAW 0,0
  120 MOVE 640,0:DRAW 640,1024
  130 MOVE 0,512:DRAW 1280,512
  140 CALL &980
  150 END
  160
  170 DEF PROCassemble
  180 old=&70:new=&72
  190 rows=&74:columns=&75
  200 temp=&76:temp1=&78:temprows=&7A
  210 osbyte=&FFF4
  220 number-of-bouncers=4*20
  230 number=&80
  240 FOR pass=0 TO 2 STEP 2
```

◀ From Page 59

```
250 P%=&980
260 [ OPT pass
270
280 JSR initialise
290 .main-loop
300 JSR move-bouncers
310 LDA #&81:LDX #256-113:LDY #&FF:JSR
osbyte  \Escape?
320 TYA:BEQ main-loop
330 RTS
340
350 .initialise
360 LDX #number-of-bouncers
370 .loop
380 STX number
390 LDA bouncer+9,X:AND #&80:BEQ next-
init \put on screen?
400 LDA bouncer,X:STA newdata+1:LDA bo
uncer+1,X:STA newdata+2 \data
410 LDY bouncer+13,X:LDA bouncer+12,X:
TAX:JSR convert \get address
420 LDX number
430 LDA new:STA bouncer+10,X:LDA new+1
:STA bouncer+11,X \store it
440 LDY bouncer+15,X:LDA bouncer+14,X:
TAX:JSR put
450 .next-init
460 SEC:LDA number:SBC #20:TAX
470 BNE loop
480 RTS
490
500 .move-bouncers
510 LDX #number-of-bouncers
520 .loop
530 STX number
540 LDA bouncer+9,X:AND #&80:BEQ next-
bouncer \on screen?
550 CLC:TXA:ADC bouncer+8,X:TAY \point
 to data
560 LDA bouncer,Y:STA olddata+1:LDA bo
uncer+1,Y:STA olddata+2
570 CLC:LDA #2:ADC bouncer+8,X:AND #7:
STA bouncer+8,X \next frame
580 CLC:TXA:ADC bouncer+8,X:TAY \point
 to data
590 LDA bouncer,Y:STA newdata+1:LDA bo
uncer+1,Y:STA newdata+2
600 LDA bouncer+10,X:STA old:LDA bounc
er+11,X:STA old+1 \old address
610 JSR get-new-xy
620 LDY bouncer+13,X:LDA bouncer+12,X:
TAX:JSR convert \get address
630 LDX number
640 LDA new:STA bouncer+10,X:LDA new+1
:STA bouncer+11,X \store it
650 LDY bouncer+15,X:LDA bouncer+14,X:
TAX:JSR print
660 .next-bouncer
670 SEC:LDA number:SBC #20:TAX
680 BNE loop
690 CLI
700 RTS
710
720 .get-new-xy
730 LDA bouncer+9,X:AND #8:BEQ leftrig
ht \going up-down?
740 LDA bouncer+9,X:AND #4:BEQ down \w
hich?
750 DEC bouncer+13,X:DEC bouncer+13,X
\y=y-2
760 JMP ud1
770 .down
780 INC bouncer+13,X:INC bouncer+13,X
\y=y+2
790 .ud1
800 LDA bouncer+13,X
810 CMP bouncer+18,X:BEQ ud2:CMP bounc
er+19,X:BNE leftright
820 .ud2
830 LDA bouncer+9,X:EOR #4:STA bouncer
+9,X \reverse y direction
840 .leftright
850 LDA bouncer+9,X:AND #2:BEQ exit-ge
t-xy \moving left-right?
860 LDA bouncer+9,X:AND #1:BEQ right \
which?
870 DEC bouncer+12,X \x=x-1
880 JMP LR1
890 .right
900 INC bouncer+12,X \x=x+1
910 .LR1
920 LDA bouncer+12,X
930 CMP bouncer+16,X:BEQ LR2:CMP bounc
er+17,X:BNE exit-get-xy
940 .LR2
950 LDA bouncer+9,X:EOR #1:STA bouncer
+9,X \reverse direction
960 .exit-get-xy
970 RTS
980
```



```
990 .put
1000 LDA #&80:STA old:STA old+1
1010 .print  \uses new/old/X=rows/Y=col
umns/olddata/newdata
1020 STX columns:STY rows
1030 LDX #0:LDY #0
1040 LDA new:STA temp1:LDA new+1:STA te
mp1+1
1050 LDA old:STA temp:LDA old+1:STA tem
p+1  \save address of column
1060 .loop1
1070 LDA rows:STA temprows
1080 .loop2
1090 .newdata LDA &3000,X:EOR (new),Y:S
TA (new),Y
1100 .olddata LDA &3000,X:EOR (old),Y:S
TA (old),Y
1110 INX
1120 LDA old:AND #7:CMP #7:BEQ bottom1
1130 INC old:BNE next1:INC old+1:JMP ne
xt1
1140 .bottom1 \row
1150 CLC:LDA old:ADC #&79:STA old:LDA o
ld+1:ADC #2:STA old+1
1160 .next1
1170 LDA new:AND #7:CMP #7:BEQ bottom2
1180 INC new;BNE next2:INC new+1:JMP ne
xt2
1190 .bottom2
1200 CLC:LDA new:ADC #&79:STA new:LDA n
ew+1:ADC #2:STA new+1
1210 .next2
1220 DEC temprows:BNE loop2  \next row
1230 CLC:LDA temp1:ADC #8:STA new:STA t
emp1:LDA temp1+1:ADC #0:STA new+1:STA te
mp1+1
1240 LDA temp:ADC #8:STA old:STA temp:L
DA temp+1:ADC #0:STA old+1:STA temp+1
1250 DEC columns:BNE loop1  \next colum
n
1260 RTS
1270
1280 .convert   \X,Y -> address in new
1290 LDA #0:STA new+1:TXA:ASL A:ASL A:R
OL new+1:ASL A:ROL new+1:STA new \X*8
1300 TYA:AND #7:ADC new:STA new:LDA new
+1:ADC #0:STA new+1 \ +(Y MOD 8)
1310 TYA:LSR A:LSR A:LSR A:ASL A:TAY  \
2*(Y DIV 8)
1320 LDA table,Y:ADC new:STA new:LDA ta
ble+1,Y:ADC new+1:STA new+1
1330 RTS
1340
1350 .table
1360 OPT FNtable
1370 ]
1380 NEXT
1390 ENDPROC
1400
1410 DEF PROCbouncer-table
1420 bouncer=&900:P%=bouncer+20
1430 RESTORE 1590
1440 FOR I%=1 TO 4
1450 READ bdata0,bdata1,bdata2,bdata3
1460 !P%=bdata0:P%!2=bdata1:P%!4=bdata2
:P%!6=bdata3
1470 READ bframe,bflags
1480 P%?8=bframe:P%?9=bflags
1490 READ baddress,bx,by
1500 P%!10=baddress:P%?12=bx:P%?13=by
1510 READ bcols,brows
1520 P%?14=bcols:P%?15=brows
1530 READ bmaxx,bminx,bmaxy,bminy
1540 P%?16=bmaxx:P%?17=bminx:P%?18=bmax
y:P%?19=bminy
1550 P%=P%+20
1560 NEXT
1570 ENDPROC
1580
1590 DATA &C00,&C00+48,&C00+2*48,&C00+3
*48,0,&8A,0,18,40,4,12,35,1,114,2
1600 DATA &C00,&C00+48,&C00+2*48,&C00+3
*48,0,&8C,0,56,8,4,12,75,31,114,2
1610 DATA &C00,&C00+48,&C00+2*48,&C00+3
*48,0,&83,0,8,184,4,12,35,1,200,128
1620 DATA &C00,&C00+48,&C00+2*48,&C00+3
*48,0,&80,0,56,180,4,12,0,0,0,0
1630
1640 DEF FNtable
1650 FOR I%=0 TO 31
1660 ?P%=(&3000+I%*&280)MOD256
1670 P%?1=(&3000+I%*&280)DIV256
1680 P%=P%+2
1690 NEXT
1700 =pass
1710
1720 REM BALL0
1730 REM X=4/Y=12
1740 DATA 0,1,1,3,15,7,7,15,3,1,1,0,3,3
,3,3,15,15,15,15,3,3,3,3,7,3,3,3,15,7,7,
15,3,3,3,7,0,2,2,3,15,15,15,15,3,2,2,0
1750 REM BALL1
1760 REM X=4/Y=12
1770 DATA 0,1,1,3,15,11,11,15,3,1,1,0,3
,3,3,3,15,15,15,15,3,3,3,3,11,3,3,3,15,1
1,11,15,3,3,3,11,0,2,2,3,15,15,15,15,3,2
,2,0
1780 REM BALL2
1790 REM X=4/Y=12
1800 DATA 0,1,1,3,15,15,15,15,3,1,1,0,7
,3,3,3,15,7,7,15,3,3,3,7,3,3,3,3,15,15,1
5,15,3,3,3,3,0,2,2,3,15,7,7,15,3,2,2,0
1810 REM BALL3
1820 REM X=4/Y=12
1830 DATA 0,1,1,3,15,15,15,15,3,1,1,0,1
1,3,3,3,15,11,11,15,3,3,3,11,3,3,3,3,15,
15,15,15,3,3,3,3,0,2,2,3,15,11,11,15,3,2
,2,0
```

This listing is included in this month's tape & disc offer. See order form on Page 167.